Simplified PageRank Algorithm Documentation

Describe the data structure you used to implement the graph and why?

To implement the graph in my program, I used an adjacency list that stored a "to" URL as the key to a map and set its value to be a vector that stores pairs. The first value of the pair is the "from" URL and the second value of the pair stores the number equal to (1 / outdegree value of the "from" URL). Using a map to implement the graph is helpful because it allows for easier access of data and avoids increasing the space complexity of storing all the URLs (compared to something like using an array instead):

```
std::map<int, std::vector<std::pair<int, double>>> graph;
```

To supplement the adjacency list, I also used three other maps to store important data used to calculate the final page rank for each URL:

1. The first map stores a string as a key and an integer as a value. This map creates a unique integer ID for each URL read by the user. This makes it easier to identify URL's without having to deal with very long strings:

   ```
   std::map<std::string, int> webpages;
   ```

2. The second map stores a string as a key and an integer as a value. This map is used to store the URLs with their respective outdegree values. Using a map to store these values allows for an O(1) time complexity when accessing a specific URL's outdegree value for computational purposes:

   ```
   std::map<std::string, int> out_degree_map;
   ```

3. The third map stores an integer as a key and a double as a value. This map is used to store the page rank respective to a URL's unique ID. Implementing a map in this case is great when conducting calculations, as accessing and manipulating the page rank for a specific URL is very quick to accomplish:

   ```
   std::map<int, double> power_iteration_map;
   ```

What is the computational complexity of each method in your implementation in the worst case in terms of Big O notation?

For these methods, the time complexity in the worst case is O (1). This is because all these functions do is return a specific map back to where they were called, which takes constant time to do. The space complexity of all these functions is also O (1), because we do not allocate any new memory.

- `std::map<std::string, int>& GetWebpages();`
- `std::map<std::string, int>& GetOutDegreeMap();`
- `std::map<int, double>& GetPowerIterationMap();`

For these methods, the time complexity in the worst case is O (log V), where V is equal to the number of vectors in the maps. This is because these functions add a key and value to a map and, since maps are sorted, the use of the [] operator for maps means that it must search for

where to place the key/value that the user inputs. The space complexity for both functions is O (1), since we are only incrementing the map by one every time the function is called.

- `void SetWebpages(std::string w);`
- `void SetOutDegreeMap(std::string o);`

For these methods, the time complexity in the worst case is O (V), where V is the number of vectors that are in the maps. Both maps use an iterator to traverse through their respective maps, which means that they are required to iterate through every key. Assuming the number of vectors stored in those maps are = V, this means that it would take linear time to get to all those vectors individually. The space complexity of these functions are O (1), as these functions do not allocate new space and are only used to print out to the user.

- `void PrintPageRank();`
- `void PrintWebpages();`

For this function, the time complexity in the worst case is O (EV^2), where V is the number of vectors and E is the number of edges. This is because the function first executes a for loop that traverses the graph, taking O (V) time. Then within that for loop, there is another for loop which traverses through the values of each key, which contain E values (worst case) and takes O (E) time to complete. Finally, within the second for loop, there is another for loop that iterates through a different map, which contains V elements and would take O (V) time. With this in mind, the final time complexity would be O (V * V * E) or O (EV^2). The space complexity for this function is O (1) as we are not creating any new memory, but rather we are modifying an existing memory's value.

- `void InitializeGraphValues();`

For this function, the time complexity in the worst case is O ((EV^2) + (PVE)), where V is the number of vectors, E is the number of edges, and P is the power iteration. We get the EV^2 because within this function, we are running InitializeGraphValues(); which has a time complexity of O (EV^2). Throughout the rest of the function, it first uses a for loop that iterates one less than the power iteration value. Within this for loop, it traverses through the graph which contains V elements in it. Finally, within that for loop, there is a third for loop that traverses the values stored within each key of the graph, which is E amount. This means that this triple for loop has a complexity of O (PEV), making the entire functions time complexity O ((EV^2) + (PVE)). The space complexity for this function is O (1) as we are not creating any new memory, but rather we are modifying an existing memory's value.

- `void PageRank(int power_iterations);`

For this method, the time complexity in the worst case is O (log V), where V is equal to the number of vectors in the map. This is because this function adds a key and value to a map and, since maps are sorted, the use of the [] operator for maps means that it must search for where to place the key/value that the user inputs. The space complexity for this function is O (1), since we are only incrementing the map by one every time the function is called.

- `void insertEdge(int from, int to);`

What is the computational complexity of your main method in your implementation in the worst case in terms of Big O notation?

Since we are using the functions previously mentioned in the main method, we can calculate complexities through addition/multiplication of those functions. The time complexity of the main method is O ((NEV^2) + (EV^2) + (PVE)), where V is the number of vectors, E is the number of edges, P is the power iteration, and N is the value given by the user and stored in "no_of_lines". We can deduce this because the main method enters a for loop that iterates N times, and from all the functions in the for loop, "insertEdge" has the largest time complexity. So we multiply N by the time complexity of "insertEdge". Then we just add this time complexity to te complexity of "PageRank" and we get the final main method time complexity. The space complexity of the main method is O (N) because we are iterating N times over the functions which all have a space complexity of O (1).

What did you learn from this assignment and what would you do differently if you had to start over?

From this assignment I have learned:

- How to design a graph through an adjacency list using maps, vectors, and pairs.
- The use of a graph and why it is so important for different scenarios.
- How to iterate through maps using the iterator function.
- More about what pairs are, why they are used, and what functions they have (never used them before this assignment).
- How to properly visualize an adjacency list and what its complexities are.

What I would do differently:

- Start the project earlier, although I did start it much earlier than project 1.
- I first used a BFS to access the graph data, to eventually realize that it wouldn't work when it came to vectors that couldn't be accessed by other vectors. So I should have thought a little bit more on how to traverse the graph before I implemented the BFS, which took a lot of time for me to do.
- I would try to find a way to minimize the number of maps I used, since I feel like I used too many maps to save data, when I could've probably used less.
- Probably should have used the debugger more; spent too much time using "cout<<" to find out what the issue with my program was, instead of properly learning how to use the debugger which could've saved me more time.
- If I had more time, I would have found out a more optimal way to iterate through a graph without having the time complexity be so large.